**First Program:**

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace MyFirstProject  {  internal class Program  {  static void Main(string[] args)  {  Console.WriteLine("Hello anand");  Console.WriteLine();  Integers integers = new Integers();  integers.integerDemo();  }  }  } |

**Let’s start from the namespace concept:**

1. **What is the namespace and why we need this...?**

A namespace in C# is a way to organize and group related classes, interfaces, structs, enums and delegates under a single name.

Eg: Thing of it like a folder in your computer where you keep related file together

So you can use the same class name twice using the different namespaces. If you want to refer any specific class then you have to use the namespace to refer the exact class which you want

**The Reason Behind the Namespace Concept:**

Namespaces were introduced to partition the space of names so the compiler (and developer) can distinguish between them.

**[**Namespace exist in C# to avoid name collisions, organize code logically, and allow large applications and multiple libraries to work together without confusion.**]**

**Now we will move forward to the next line [ internal class MyFirstProject ] what does the line is all about:**

Internal is the keyword for accessing the class within the assembly.   
let’s talk about the access what it means…? To accessing class or method/ property /field we need to have an access for that.

**How do we get the access…?**

In C# (or any other programming language) there is a concept called access specifiers these access specifiers have different meanings for different keywords so each one will give you a specific range of accessing a class or method/property/field.

**So, the how many types of access modifiers are available in C#...?**

C# have the 6 different access modifiers

|  |  |
| --- | --- |
| **Modifier** | **Accessible From** |
| public | Anywhere (no restriction) |
| private | Inside the same class only |
| protected | Inside the class and its subclasses |
| internal | Anywhere in the same assembly (same project) |
| Protected internal | Subclasses + same assembly |
| Private protected | Subclasses but only if in the same assembly |

Internal class MyFirstProject means the class can only be used inside the same project/assembly where it is defined, but it won’t be accessible from outside projects.

**Doubt:**

Do we have to use the same name for the namespace and the class..?

No, you don’t need to use the same name for the namespace and the class because name space is used to organize and group the multiple classes in the same project so we need to use the different names for the class and namespace. Even if you keep the same name for the class and name space there is no issue but it makes you confuse so better to use different names for the namespace and the class.

**Why we need to include the all the namespaces before using it…?**

If you don’t included the name space system.io then you have to indicate the Console.WriteLine() function is belongs to which namespace like   
  
System.Console.WriteLine();

If you include then you don’t need to write the System Infront of every Console.WriteLine()

Variable declaration:

In C# you can declare a variable like declaring which data type it is and the name of the variable.

Syntax: int age; //This is how we can declare a integer type variable

Initializing a variable:

To initialize the variable, we need to use the variable name with the assigning operator along with the value of the variable (must be same as the declared datatype).

Syntax: age = 25;

Question  
Can we declare a variable and assign the value in one line…?

Yes, it’s possible to declare and assign a value to the variable in one line

Syntax: int my\_age = 25; // declaring and initializing the variable

**What are the different data types are available in the C#:**

There are mainly 3 categories of data types:

* Value Types
* Reference Types
* Pointer Types (unsafe-code)

**1. Value Types:**

* Store the data directly in memory
* Stores in the stack
* Examples: numbers, Booleans, structs, enums.

| **Data Type** | **Size** | **Description** | **Example** |
| --- | --- | --- | --- |
| bool | 1 byte | True/False values | bool isActive = true; |
| byte | 1 byte | 0 to 255 | byte b = 200; |
| sbyte | 1 byte | -128 to 127 | sbyte sb = -50; |
| char | 2 bytes | A single Unicode character | char c = 'A'; |
| short | 2 bytes | -32,768 to 32,767 | short s = -200; |
| ushort | 2 bytes | 0 to 65,535 | ushort us = 40000; |
| int | 4 bytes | -2.1B to 2.1B | int num = 100; |
| uint | 4 bytes | 0 to 4.2B | uint u = 300000; |
| long | 8 bytes | Very large integers | long big = 922337203685; |
| ulong | 8 bytes | Very large positive integers | ulong ul = 9999999999; |
| float | 4 bytes | Single precision decimal (7 digits) | float f = 3.14f; |
| double | 8 bytes | Double precision decimal (15-16 digits) | double d = 3.14159; |
| decimal | 16 bytes | High precision (28-29 digits), for money | decimal price = 199.99m; |

**2. Reference Types**

* Store the reference (address) of the data, not the actual data.
* Stores in the heap (but reference itself is in the stack).
* Example: object, arrays, strings.

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| object | Base type of all types in C# | object obj = 42; |
| string | Sequence of characters | string name = "Anand"; |
| dynamic | Type checking at runtime | dynamic x = 10; x = "Hello"; |
| var | Type inferred at compile time | var age = 25; |

User-defined Reference Types

* Class – blueprint for objects
* Interface – contract without implementation
* Delegate – reference to a method

**3. Pointer Types (unsafe code)**

* Rarely used, mostly in low-level programming
* Store the memory address of the another variable
* Requires unsafe keyword.

|  |
| --- |
| usafe{  int x = 10;  int \*p = &x;    // \* is used to refer the pointer variable and & is used to refer the address of the variable  Console.WriteLine(\*p);  // prints the 10  } |

We have one more type

**Special Type: Nullable**

**Allow value types to hold null (null value is a value which does not have any value)**

**Summary:**

* **Value Types:** int, float, bool, struct, enum (stored in stack)
* **Reference Types:** string, object, class, interface, delegate, array (stored in heap)
* **Pointer Types:** Direct memory access (unsafe)
* **Nullable Types:** Allow value types to accept null

**C#** data types are divided into **Value types (Stores the actual values), Reference types (store memory references), Pointer types (low-level memory access), and Null types (value types can hold null).**

**What is the data type conversion?**

Data type conversion is the process of converting one data type to another data type.

Some times the type of data we need is different then the type of data we have at that time we have to use the data type conversion

In C# we have 4 different types of the data type conversion are there

* Implicit Type conversion (Type Casting)
* Explicit Type conversion (Type Casting / Type Casting Operator)
* Using Conversion method
* Parsing & TryParse

**Implicit Type conversion:**

-Done automatically by the compiler

-Only works when there’s no data loss

-Usually from smaller type -> larger type

|  |
| --- |
| int a = 100;  double b = a; // int → double (safe, no data loss) |

**Explicit conversion:**

-Done manually using (type)

-Needed when there might be data loss

|  |
| --- |
| double a = 100;  int b = (int)a; //double to int (decimal part lost)  Console.WriteLine(i); |

**Using Conversion methods**

-Provided by the .NET Framework through Convert class

|  |
| --- |
| string s = “123”;  int num = Convert.ToInt32(s); string ->int  Console.WriteLine(num + 10); -> 133  Bool flag = Convert.ToBoolean(1); //1->true |

**Parsing & TryParse:**

-Used to convert string -> numeric safely.

|  |
| --- |
| string str = “abc”;  int value = iint.Parse(str); //works if string is valid number  Console.WriteLine(value+50); //250 |

* But if the string is not a number, Parse() throws an exception.
* So we use TryParse:

|  |
| --- |
| string str = “abc”;  bool success = iint.TryParse(str, out int result);  if (success)  Console.WriteLine(result);  else  Console.WriteLine(“Invalid Number”); |

| **Method** | **Example** | **Risk** |
| --- | --- | --- |
| **Implicit Conversion** | int → double | Safe |
| **Explicit Casting** | (int)3.14 | Data loss |
| **Convert Class** | Convert.ToInt32("123") | Exception if invalid |
| **Parse** | int.Parse("123") | Exception if invalid |
| **TryParse** | int.TryParse("123", out x) | Safe, returns false if invalid |

So the answer is yes we can convert the data types in C# using implicit casting, explicit casting, Convert class, Parse, and TryParse, depending on safety and data loss concerns.

**Conditional statements:**

Conditional statements are used to control the flow of control of the program to how they have to execute. So it allows you to change the order of executing statements or methods or function.

* If statement
* If else statement
* Else statement
* Switch statement

There are the control statements available in the C#.

If statement:

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.linq;  using System.Text;  using System.Treading.Task;  namespace MyFirstProject  {  internal class Prgram  {  static void Main(String[] args)  {  int a = 20, b = 2, c;  if (a > b)  Console.WriteLine(“a is greater than the b”);  else if (a < b)  Console.WriteLine(“a is smaller than the b”);  else  Console.WriteLine(“a is equal to b”);  }  }  } |

**Switch statement:**

Unlike the if else statement switch works differently. If else is check the each block by validating every condition one after another but switch works based on the case which will directly execute the code if the case is match.

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace MyFirstProject  {  internal class SwitchStatement  {  public void SwitchStatementcheck()  {  Console.Write("Enter the day: ");  int day = Convert.ToInt32(Console.ReadLine());  switch (day)  {  case 1:  Console.WriteLine("Monday");  break;  case 2:  Console.WriteLine("Tuesday");  break;  case 3:  Console.WriteLine("Wednessday");  break;  case 4:  Console.WriteLine("Thursday");  break;  case 5:  Console.WriteLine("Friday");  break;  case 6:  Console.WriteLine("Saturday");  break;  case 7:  Console.WriteLine("Sunday");  break;  }  }  }  } |

# **C# – Short Notes**

## 🔹 1. Basics

* Case-sensitive, strongly typed, object-oriented.
* Entry point:
* static void Main(string[] args) { }

## 🔹 2. Data Types

* **Value Types**: int, float, double, decimal, bool, char, struct
* **Reference Types**: string, object, array, class, interface
* **Nullable Types**: int? x = null;
* **Var / Dynamic**:
  + var → type decided at compile time.
  + dynamic → type checked at runtime.

## 🔹 3. Variables & Constants

int a = 10;

var b = 20; // type inferred

const double PI = 3.14;

readonly int id; // set in constructor

## 🔹 4. Operators

* Arithmetic: + - \* / %
* Comparison: == != > < >= <=
* Logical: && || !
* Null-coalescing: ??
* Ternary: condition ? value1 : value2

## 🔹 5. Control Flow

**If-Else**

if(x>0) Console.WriteLine("Positive");

**Switch**

* Old: constants only.
* Modern (C# 7+): patterns + when.
* Expression (C# 8+): concise.

## 🔹 6. Loops

for(int i=0; i<5; i++) { }

while(cond) { }

do { } while(cond);

foreach(var n in nums) { }

* break → exit loop.
* continue → skip iteration.

## 🔹 7. Methods

int Add(int x, int y) => x + y;

* **Overloading**: same name, different parameters.
* **Parameters**:
  + ref (pass by reference)
  + out (output only)
  + params (variable args)

## 🔹 8. OOP Concepts

* **Class** → blueprint.
* **Object** → instance.
* **Encapsulation** → private fields + public properties.
* **Inheritance** → class Dog : Animal { }
* **Polymorphism** → method overloading/overriding.
* **Abstraction** → abstract class, interface.

## 🔹 9. Properties

public string Name { get; set; } // Auto-property

* Can have **get/set** logic.

## 🔹 10. Exception Handling

try { }

catch(Exception ex) { }

finally { }

throw new Exception("Error");

## 🔹 11. Collections

* **Array**: fixed size.
* **List**: dynamic size.
* **Dictionary<TKey,TValue>**: key-value pairs.
* **Queue / Stack**: FIFO / LIFO.

## 🔹 12. LINQ

var result = nums.Where(n => n > 5).Select(n => n\*2);

* Query syntax also supported:

var result = from n in nums

where n > 5

select n\*2;

## 🔹 13. Async & Await

async Task<int> GetData()

{

await Task.Delay(1000);

return 42;

}

## 🔹 14. File I/O

File.WriteAllText("file.txt", "Hello");

string text = File.ReadAllText("file.txt");

## 🔹 15. Keywords Quick Recap

* this → current object.
* base → parent class.
* static → shared across objects.
* sealed → class can’t be inherited.
* override → override base method.
* virtual → allow override.
* abstract → must override.
* using → import namespace / manage resources.

**C# Handbook – Mindmap (Outline)**

C#

|

|── Basics

│ |── Main() entry point

│ |── Case-sensitive, OOP

│

|── Data Types

│ |── Value → int, float, double, bool, char, struct

│ |── Reference → string, object, array, class

│ |── Nullable → int? x = null

│ |── var (compile-time) vs dynamic (runtime)

│

|── Variables

│ |── Normal → int x = 10;

│ |── const → compile-time constant

│ |── readonly → set in constructor

│

|── Operators

│ |── Arithmetic (+ - \* / %)

│ |── Comparison (== != > <)

│ |── Logical (&& || !)

│ |── Ternary (? :) (Conditional operator)

│ └── Null-coalescing (??)

│

|── Control Flow

│ |── if-else

│ |── switch

│ │ |── Classic (constants)

│ │ |── Pattern matching (when, C#7+)

│ │ └── Switch expression (C#8+)

│

|── Loops

│ |── for

│ |── while

│ |── do-while

│ └── foreach

│

|── Methods

│ |── Normal & Expression-bodied

│ |── Overloading

│ |── Parameters → ref, out, params

│

|-─ OOP Concepts

│ |── Class & Object

│ |── Encapsulation → private + properties

│ |── Inheritance → `:`

│ |── Polymorphism → overloading/overriding

│ └── Abstraction → abstract class, interface

│

|── Properties

│ └── Auto-property { get; set; }

│

|── Exception Handling

│ |── try-catch-finally

│ └── throw new Exception()

│

|── Collections

│ |── Array

│ |── List<T>

│ |── Dictionary<TKey,TValue>

│ |── Queue (FIFO)

│ └── Stack (LIFO)

│

|── LINQ

│ |── Method syntax → Where(), Select()

│ └── Query syntax → from ... where ... select

│

├── Async/Await

│ └── async Task, await

│

|── File I/O

│ |── File.WriteAllText()

│ └── File.ReadAllText()

│

└── Keywords

|── this / base

|── static / sealed

|── virtual / override / abstract

└── using (namespace & resource mgmt)

## What is the Conditional (Ternary) Operator?

The conditional operator ?: is a **short form of** if...else.

|  |
| --- |
| condition ? expression\_if\_true : expression\_if\_false |

|  |
| --- |
| int age = 20;  string result = (age >= 18) ? "Adult" : "Minor";  Console.WriteLine(result); // Adult |

## Example 3: Nested Conditional Operator

|  |
| --- |
| int marks = 85;  string grade = (marks >= 90) ? "A" :  (marks >= 75) ? "B" :  (marks >= 50) ? "C" : "Fail";  Console.WriteLine(grade); // B |